ENGR-UH 3511

Computer Organization and Architecture

Name: Nishant Aswani

Net ID: nsa325

Assignment Title: Lab 1

Introduction to the Bash Shell and Performance Benchmarks

Nishant Aswani, nsa325@nyu.edu

Computer Organization and Architecture(ENGR-UH 3511), Instructor Cristoforos Vasilatos

1 Introduction

Performance benchmarks are used to determine how well a machine compares against other machines. Using the same benchmark between machines provides a relative measurement of how well certain hardware performs during a certain task. This assignment focuses on the LINPACK performance benchmark, which solves a "dense system of linear equations" [1].

2 Methodology

The LINPACK benchmark script was downloaded from the netlib repository [2]. Then, the program was compiled thrice, with each compilation carrying out varying levels of optimization. The first compilation had no optimizations, the second compilation used the `-O1` flag, and the third compilation used the `-O2` flag. The `perf stat` command was run on each executable, followed by the `perf record` and `perf report` commands. Each program used an array size of 200.

The full benchmark and perf stat results for each iteration can be found in the appendix.

Looking into the results of the `perf report` command, the top five instructions from a Symbol related to a function of the benchmark were recorded. Data was also retrieved from `perf record` and `perf stat` to calculate the total clock cycles of the relevant instructions. The results from the three iterations of the benchmark were tabulated. The print functionality in `perf report` was used to store a text copy of the annotations.

3 Results

3.1 No Optimizations

The following commands were run:

```
gcc -o noOptimization linpack.c -lm
sudo perf stat ./noOptimization
```

The program terminated with `perf stat` statistics, which included the following information of interest:

```
122,796,640,077 cycles       # 3.076 GHz
366,548,028,130 instructions # 2.99  insn per cycle
```

Next, the `perf record` and `perf report` commands were run. At 40.07%, the daxpy_r symbol had the highest overhead. The following were the five instructions with the highest overhead percentage.

```
16.96%   addsd  %xmm1,%xmm0
11.19%   movsd  %xmm0,(%rax)
11.15%   jl     10a
11.01%   mov    -0x30(%rbp),%rax
10.04%   movsd  (%rax),%xmm0
```

From the perf wiki, we learn that the "by default, perf record uses the cycles event as the sampling event." [3]

Thus, assuming that the daxpy_r is 40.07% of the total cycle count, we can calculate the cycle count for the function.

$$\begin{aligned} \text{CC} &= 122{,}796{,}640{,}077 \cdot 0.4007 \\ &= 49{,}204{,}613{,}679 \text{ cycles} \end{aligned} \tag{1}$$

The top five instructions can be calculated as a percentage of this value, as the percentages are all local. The results are tabulated in Table 1.

Cycle Count	%	Instruction
8,345,102,480	16.96	addsd %xmm1,%xmm0
5,505,996,271	11.19	movsd %xmm0,(%rax)
5,486,314,425	11.15	jl 10a
5,417,427,966	11.01	mov -0x30(%rbp),%rax
4,940,143,213	10.04	movsd (%rax),%xmm0

Figure 1: Cycle counts of the top 5 instructions in daxpy_r with no optimizations

We can confirm these values by using the global percentage provided by `perf report` and multiplying with the total cycle count. The global percentage for the hottest instruction is 6.80%. Equation 2 attempts to calculate the cycle count using another method:

$$\begin{aligned} \text{CC} &= 122{,}796{,}640{,}077 \cdot 0.0680 \\ &= 8{,}350{,}171{,}525 \text{ cycles} \end{aligned} \tag{2}$$

Seeing the similarity in values obtained from both methods confirms the idea that the overhead percentage uses cycles as events.

3.2 Optimization Level One

The following commands were run:

```
gcc -o optimizationLevelOne -O1
linpack.c -lm

sudo perf stat ./optimizationLevelOne
```

These were some of the following `perf stat` statistics of interest:

```
70,893,522,712  cycles       # 3.078 GHz
234,689,115,718 instructions # 3.31  insn
                             per cycle
```

The daxpy_r symbol, with an overhead of 28.68%, was selected and the five instructions with most overhead were as follows:

```
30.50%   mulsd  (%rsi,%rax,1),%xmm1
29.86%   movsd  %xmm1,(%rcx,%rax,1)
23.21%   jne    44
6.40%    addsd  (%rcx,%rax,1),%xmm1
1.37%    mov    $0x0,%eax
```

As before, the cycle count for the selected function was calculated using Equation 3:

$$\begin{aligned} \text{CC} &= 70{,}893{,}522{,}712 \cdot 0.2868 \\ &= 20{,}332{,}262{,}314 \text{ cycles} \end{aligned} \tag{3}$$

The cycle counts and local percentages are tabulated in Table 3

Cycle Count	%	Instruction
6,201,340,005	30.50	mulsd (%rsi,%rax,1),%xmm1
6,071,213,527	29.86	movsd %xmm1,(%rcx,%rax,1)
4,719,118,083	23.21	jne 44
1,301,264,788	6.40	addsd (%rcx,%rax,1),%xmm1
2,78,551,994	1.37	mov $0x0,%eax

Figure 2: Cycle counts of the top 5 instructions in daxpy_r with level 1 optimizations

3.3 Optimization Level Two

The following commands were run:

```
gcc -o optimizationLevelTwo -O1
linpack.c -lm

sudo perf stat ./optimizationLevelTwo
```

These were the statistics of interest from `perf stat`:

```
70,121,964,990  cycles       # 3.068 GHz
207,965,402,026 instructions # 2.97  insn
                             per cycle
```

Once again, the daxpy_r symbol, at an overhead of 34.85%, was selected and the five instructions with most overhead were as follows:

```
33.56%   mulsd  %xmm0,%xmm1
32.94%   movsd  %xmm1,(%rdx,%rax,8)
13.24%   addsd  (%rdx,%rax,8),%xmm1
12.11%   jg     1b
3.23%    36:   repz   retq
```

The cycle count for daxpy_r was calculated using Equation 4:

$$\begin{aligned} \text{CC} &= 70,121,964,990 \cdot 0.3485 \\ &= 24,437,504,799 \text{ cycles} \end{aligned} \tag{4}$$

The derived counts and local percentages are tabulated in Table 3

Cycle Count	%	Instruction
8,201,226,610	33.56	mulsd %xmm0,%xmm1
8,049,714,081	32.94	movsd %xmm1,(%rdx,%rax,8)
3,235,525,635	13.24	addsd (%rdx,%rax,8),%xmm1
2,959,381,831	12.11	jg lb
789,331,405	3.23	36: repz retq

Figure 3: Cycle counts of the top 5 instructions in daxpy_r with level 2 optimizations

4 Discussion

Optimizations led to a decrease in the total cycle counts. There was a 42% decrease in cycle counts after using the `-O1` flag when compared to no optimizations. The `-O2` flag led to an approximately 43% decrease. There was not much of an improvement in cycle counts for the program using a second level optimization.

The optimizations also improved the instruction count. A level one optimization led to a roughly 36% decrease in the number of instructions. Further, a level two optimization led to a 43% decrease in the number of instructions. Therefore, one my conclude that the decrease in instruction count may be a more accurate representation of the optimization carried out by the compiler.

The optimizations also led to different instructions taking a higher weight and the usage of different registers. Both level one and level two optimizations carry out mulsd the most, while the unoptimized code carries out addsd the most. Mulsd refers to multiply scalar double-precision floating-point values, while addsd to refers to add scalar double-precision floating-point values. It is likely that the optimization found a way to carry out a piece of code by using multiplication, rather than addition. Looking at the cycle count for mulsd when using `-O1`, it is less than that of no optimization. However, using the `-O2` flag actually does not reduce the cycle count for the hottest instruction.

The high addsd and mulsd instructions can be explained by the function of the program, as it calculating a system of linear equations. If somehow optimization reduces the number of add operations, the higher overhead of multiplication operations may be explained by the fact that multiplication simply takes a longer time, and therefore more cycle counts.

The move instruction interacting with memory is a hot instruction prior to optimization; after optimization, the hottest move instruction does not involve interaction with memory. Hence, it may be that optimization reduces the interaction with memory and makes better use of registers.

Finally, certain instructions such as jl, jne, and jg are all conditionals, and are likely to be a manifestation of the for loop that is in the daxpy_r function in the program. It is interesting to note that each of the iterations primarily uses a different kind of jump instruction.

Overall, the optimization clearly affects the program as seen by the decrease in instructions, as well as the decrease in task clock (see Appendix). However, the difference between no optimization and `-O1` optimization is much larger than the change between `-O1` and `-O2` optimization.

References

[1] *The LINPACK Benchmark*. URL: https://www.top500.org/project/linpack/.

[2] *Netlib*. URL: http://www.netlib.org/benchmark/linpackc.new.

[3] *Tutorial*. URL: https://perf.wiki.kernel.org/index.php/Tutorial.

5 Appendix

```
No Optimization

gcc -o noOptimization linpack.c -lm
sudo perf stat ./noOptimization

LINPACK benchmark, Double precision.
Machine precision:  15 digits.
Array size 100 X 100.
Average rolled and unrolled performance:

    Reps Time(s) DGEFA   DGESL  OVERHEAD    KFLOPS
----------------------------------------------------
    2048   0.62  79.91%   5.31%  14.78%  682898.594
    4096   1.25  79.93%   5.30%  14.77%  677300.656
    8192   2.52  79.92%   5.32%  14.76%  673195.880
   16384   4.99  79.90%   5.30%  14.79%  680961.693
   32768   9.96  79.93%   5.29%  14.78%  682137.849
   65536  19.92  79.91%   5.30%  14.79%  682114.902

Enter array size (q to quit) [200]:  q

 Performance counter stats for './noOptimization':

         39,921.21 msec task-clock                #    0.926 CPUs utilized
               139      context-switches          #    0.003 K/sec
                 4      cpu-migrations            #    0.000 K/sec
                67      page-faults               #    0.002 K/sec
   122,796,640,077      cycles                    #    3.076 GHz
   366,548,028,130      instructions              #    2.99  insn per cycle
    15,293,428,879      branches                  #  383.090 M/sec
       152,336,770      branch-misses             #    1.00% of all branches

      43.132296218 seconds time elapsed

      39.697325000 seconds user
       0.224007000 seconds sys


******************************************************************

-O1 Optimization

gcc -o optimizationLevelOne -O1 linpack.c -lm
sudo perf stat ./optimizationLevelOne
```

```
LINPACK benchmark, Double precision.
Machine precision:  15 digits.
Array size 100 X 100.
Average rolled and unrolled performance:

    Reps Time(s) DGEFA   DGESL  OVERHEAD    KFLOPS
----------------------------------------------------
    8192   0.72  68.85%   5.73%  25.42%  2700865.420
   16384   1.44  68.84%   5.71%  25.46%  2701180.479
   32768   2.88  68.87%   5.71%  25.42%  2691555.321
   65536   5.75  68.83%   5.72%  25.45%  2702318.454
  131072  11.49  68.84%   5.73%  25.44%  2702859.407

Enter array size (q to quit) [200]:  q

 Performance counter stats for './optimizationLevelOne':

        23,029.94 msec task-clock                #    0.813 CPUs utilized
              141      context-switches          #    0.006 K/sec
                3      cpu-migrations            #    0.000 K/sec
               68      page-faults               #    0.003 K/sec
   70,893,522,712      cycles                    #    3.078 GHz
  234,689,115,718      instructions              #    3.31  insn per cycle
   26,551,303,639      branches                  # 1152.903 M/sec
      151,894,701      branch-misses             #    0.57% of all branches

      28.339501583 seconds time elapsed

      22.374523000 seconds user
       0.655839000 seconds sys



*****************************************************************

-O2 Optimization

gcc -o optimizationLevelTwo -O1 linpack.c -lm
sudo perf stat ./optimizationLevelTwo



LINPACK benchmark, Double precision.
Machine precision:  15 digits.
```

```
Array size 100 X 100.
Average rolled and unrolled performance:

    Reps Time(s) DGEFA   DGESL  OVERHEAD    KFLOPS
----------------------------------------------------
    8192   0.71  68.65%   5.78%  25.58%  2734660.303
   16384   1.42  68.58%   5.76%  25.67%  2732643.967
   32768   2.88  68.62%   5.78%  25.60%  2700326.209
   65536   5.75  68.66%   5.76%  25.58%  2706987.548
  131072  11.36  68.61%   5.76%  25.63%  2739614.999

Enter array size (q to quit) [200]:  q

 Performance counter stats for './optimizationLevelTwo':

         22,852.84 msec task-clock                #    0.884 CPUs utilized
               120      context-switches          #    0.005 K/sec
                 1      cpu-migrations            #    0.000 K/sec
                66      page-faults               #    0.003 K/sec
    70,121,964,990      cycles                    #    3.068 GHz
   207,965,402,026      instructions              #    2.97  insn per cycle
    23,845,508,546      branches                  # 1043.438 M/sec
       151,842,395      branch-misses             #    0.64% of all branches

      25.860533063 seconds time elapsed

      22.296726000 seconds user
       0.555918000 seconds sys
```